Event-Driven I/O
A hands-on introduction

Marc Seeger
HdM Stuttgart

August 8, 2010

Abstract

This paper gives a short introduction to the topic of asynchronous, event-driven I/O, explains the underlying "reactor" pattern and introduces the usage of lightweight threads to help with readability of event-driven code.

Contents

1 Introduction

When it comes to application development, there are a few things that separate regular single-user desktop applications from server applications. The one main thing however is, that server applications tend to have many concurrent users while desktop applications can usually be designed for a single user at any given time.
One of the most important differences is the way that the applications handles data input and output. Desktop Applications usually receive data over the network or load files from disk using "synchronous" or "blocking" I/O. This is usually the easy way and is fine for a single user. As soon as an application has to interact with several users or large amounts of input/output, using an non-blocking I/O becomes the only way of keeping the application responsive and allowing it to scale.

2 General scalability problems

When trying to create a single service that will scale for thousands of incoming requests, there are two main things that influence the performance of your application: Computation and I/O.

2.1 Computations

Computations are the part of your application that actually use most of the available CPU. They are the functional components that transform data from one form into another. This can be compressing images, translating text into other languages, sorting numbers according to size.
The main problem is that your server's CPU has only a limited amount of operations per second. Your algorithm needs a certain number of cycles for every bit of input. Once you max out this ratio, your server is at peak capacity and every added request will have to wait for the other ones to finish.
The simplest solution to this problem is adding more and faster hardware to the system in question. You can buy faster CPUs to be able to process more cycles per second. The problem is that the cost and the amount of additional scalability you can gather is very limited by the manufacturing process and clockspeeds of current CPUs. There is usually only little difference between the clock speed of a regular CPU and a high-end CPU. There is however a difference in the amount of available cores between the regular (2) and the high end (8) Processors. This leads us to the next way to solve the computational problem. A solution that is a little bit more sophisticated

is the partitioning of data by the algorithm. This allows each part of the split-up computation to be processed by a different core of the CPU. A simple assumption would be, that an 8 core CPU will now solve a given problem 4 times as fast as a dual-core CPU. This is however a bit oversimplified. If you're interested in this topic, I can recommend Kai Jäger's Bachelor Thesis "Finding parallelism - How to survive in a multi-core world" (2008) [2].
Taking the parallelization to the next step means that we distribute the problem not only between several cores of a CPU, but between several networked computers and thus have a simple way of scaling our system by adding additional machines to the network. A generalized approach to this problem has been introduced by Jeffrey Dean and Sanjay Ghemawat (both employed by Google) in a paper called "MapReduce: Simplified Data Processing on Large Clusters". If the scalability of the application is CPU bound, transforming the applications different tasks into MapReduce elements might be the best way to go. An open-source set of frameworks that (among other things) implements MapReduce and helps with coordination and network interaction between the processing nodes is called Apache Hadoop. It is widely used and worth a look if your problem is the processing of vast amounts of data on a large amount of machines/cores.

2.2 I/O

The other problem that keeps applications from scaling is Input and Output of data to attached devices. These devices can be network cards, harddiscs or anything else that isn't directly connected to the reasonably fast RAM or the CPU.
Using MapReduce helps to process a large amounts of data in a short amount of time and thus solves the computation problem, but it doesn't help with large numbers of small operations.
The single pieces of data are usually pretty easy to process and distributing the single piece of data among more than one node, sometimes even among more than one CPU core, isn't usually worth the overhead that comes with it. Most of the time between the start of the request and the result is actually being spent waiting for data to be copied in and out of buffers and checking if the response has arrived back yet. A simple, single-threaded application would completely lock down while it waits for the harddisc to read a file, the network card to collect the whole request packet, the database to answer a query or a user to push a button. Even multi-threaded applications suffer from problems when it comes to I/O. Every time an application is waiting for an I/O request using a thread, it doesn't block the rest of the application. It does however add another piece of memory to your applications footprint and

depending on the implementation, might cause a lot of unnecessary switches between threads that add a lot of overhead to the whole process.
This is why it is important to look at the way a single server process can handle incoming data without locking down completely.

3 Threads: Concurrency the old fashioned way

To understand what concurrency using threads does, one has to understand the process an application has to go through, to offer its services over the network.
An application has to attach itself to a certain TCP/UDP port. This way, the operating system will pass all incoming data that is addressed to that port to the application listening on it. To create a simple Service, one doesn't have to do much.
As a little example, here is a little server that does echo back everything that is being sent to it:

```
require 'socket'
server = TCPServer.new(2803)
while client = server.accept
    input = client.readline
    client.write "You said: #{input}"
    client.close
end
```

This application will run to the server.accept call which is "blocking". This means that the application will stop until somebody actually opens up a connection to the specified port (2803 in this case). This application does a pretty good job as long as there is only one user at a time. As soon as 2 users want to connect to the server at the same time, one of them will have to wait for the server to message back to the first user.
The problem in the simple solution is, that everything that happens between "server.accept" and "end" will keep our server from servicing a second connection. This is where Threads come in.
Since a thread usually returns immediately and does its work somewhere in the background, one can simply but everything between "server.accept" and "end" into a thread, spawn it off and immediately start working on the next connection that came in.
The modifications to our application are minimal:

```
require 'socket'
server = TCPServer.new(2803)
loop do
    Thread.new(server.accept){ |client|
        input = client.readline
        client.write "You said: #{input}"
        client.close
    }
end
```

The only thing one has to pay attention to is, that the application has to be "thread-safe" now. This means that whatever data your application changes has to be secured by some kind of locking. Otherwise, two threads that run in parallel would be able to produce inconsistent data in the backend or end up in a race-condition. In our example, we don't change any data besides the output that gets echoed back to the user. All of the "state" of the thread is embedded in the thread itself. This is comparable to the main ideas behind functional programming and allows us to run these threads without any locking backend data.
This approach isn't without problems. For each request, a new thread has to be spawned up, kept in memory and managed by the operating system. This leads to a situation in which you have a 1:1 ratio between threads and users and are limited by the ability of your operating system to balance these threads and the RAM it takes to keep them in memory. While this allows a server that only does lightweight operations in the threads (e.g. an HTTP server) to scale for a while, at a few thousand concurrent connections, the sheer management of the threads becomes a problem and the performance drops.

4 Reactor and Proactor

There are two design patterns that try to solve the problem of having a 1:1 ratio between threads and users. Reactor and proactor are both I/O multiplexers. They form a central instance and allow a programmer to register event-handlers for specific I/O events. They employ an event demultiplexer that calls the registered handlers whenever the event occurs. This means that the programmer doesn't have to keep all of his threads in memory at the same time. The application logic for dealing with the I/O responses is executed when the response is actually there and not beforehand. This keeps the amount of switching between threads and blocking for results close to

zero.
The difference between reactor and proactor are the events that are being monitored.
In the (synchronous) reactor pattern, the programmer registers callbacks on "ready to read/write" events. Whenever one of these events happens, the callback is being invoked and the application logic reads from or writes to the available file descriptor. It has to be noted that file descriptors in Unix can either be an actual file on the file system or a representation of a network-socket.
In the (asynchronous) proactor pattern, the programmer starts an asynchronous write/read operation by executing the operation-system's asynchronous read/write call and passing it a user-defined buffer. Additionally, a callback on a "write/read complete" event is set. While the event demultiplexer waits for changes, the operating system launches a parallel kernel thread that does the actual write/read operation and copies the data from/to the user-defined buffer. Once the operation is complete, the "complete"-event gets fired and the invoked callback can continue writing/reading additional data with another systemcall. In conclusion: The main difference between reactor and proactor is the way they do the actual write/read. While the reactor is only being signalled when the socket is ready to write/read and then does the reading/writing itself in a synchronous manor, the proactor will use an asynchronous I/O call to the operating system and only supply the buffers to read from/write to.
A more detailed comparison between the two can be found in "Comparing Two High-Performance I/O Design Patterns" by Alexander Libman and Vladimir Gilbourd [1]

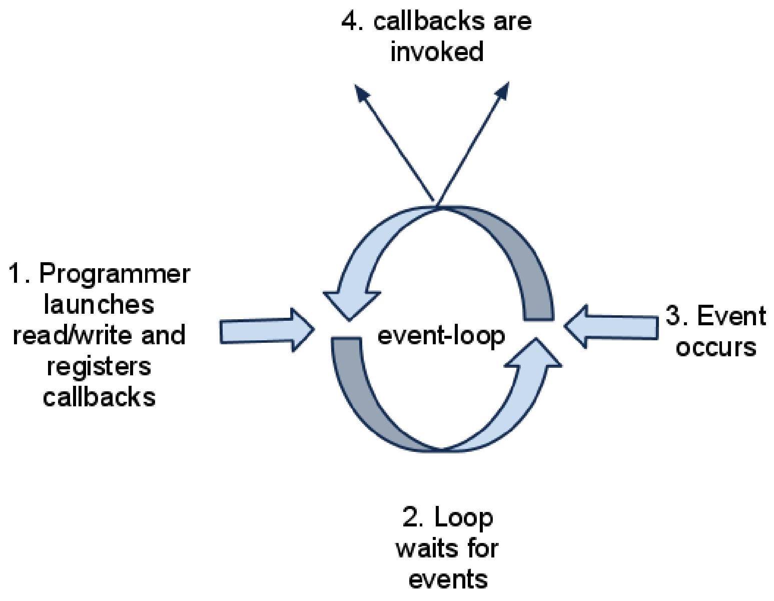

Figure 1: event-loop

4.1 Working with the reactor

When working with the reactor pattern, there are a few things that one has to pay attention to.
The most important one is: Don't stop the reactor! Most frameworks that provide a generic implementation of the reactor pattern are single-threaded. This makes perfect sense as the thread overhead is exactly what we wanted to stay away from. This also means that the reactor loop will be executing your callbacks and resume once they finished. Making your callback code as lightweight as possible is a best practice when working with this pattern.
This means mainly that these things should be avoided:

- loops that iterate over large collections of objects
- computation intensive things
- calling synchronous I/O methods (printing to stdout might also be part of this depending on its implementation)
- sleep/pause statements

If there really is a need for these operations, they should be done within an extra thread.

5 Fighting complexity - Lightweight threads as syntactic sugar

While the reactor pattern offers a great solution to I/O problems when trying to scale, it is definitely not a "drop-in" solution. Using the pattern requires evented code to be registered as callbacks.
This code is in almost all cases more complex (and bigger) than its 'regular' counterpart. Especially things like nested callbacks require a lot of concentration and are lacking readability.
While this certainly is a problem, some programming languages are able to help by adding a lightweight alternative to threads to the mix which are usually known as "Coroutines".
These lightweight threads are e.g. "Coroutines" in Lua, "Fibers" in Ruby, "tasklets" in stackless Python or in general: lightweight constructs that are able to "yield" and "resume" at a given time.
An overview of programming languages supporting this construct can be found on the "Coroutine" page on Wikipedia[1].
A good example of this can be found in Ilya Grigorik's wonderful post blogpost on "Untangling evented code with ruby fibers"[3]. He shows how wrapping our callback functions in Ruby Fibers can help bring some readability back to our code and still keep it asynchonous at heart. By wrapping them, we can define them as a "regular" function outside of our reactor-loop and still call them from within the loop without the risk of accidentally blocking the reactor. Since Fibers, Ruby's lightweight concurrency-elements, are basically user-level Threads without preemption. They have to be yielded and resumed by hand. This allows us to do the following workflow:

1. set up asynchronous call
2. add a callback that will resume the Fiber once the call it is done
3. immediately yield the Fiber

To give a better example, here is a generic method that would allow us to call it from within the reactor with minimal overhead, thus keeping the reactor loop small. The difference to a "regular" function call is, that this piece of code will almost immediately return and NOT block the reactor.

[1]http://en.wikipedia.org/wiki/Coroutine

```
def do_something(parameter)
  # get the ID of our current Fiber
  f = Fiber.current
  # set up an asyncronous call
  my_async_call = Library.do_something()

  # register callback to resume fiber
  # once the async call is done
  result = my_async_call.callback { f.resume() }

  # give control back to the main Process
  Fiber.yield

  # This is where we’ll return to after after the fiber
  # has been resumed by the callback. Our "result"
  # variable should be set and we can return it
  return result
end
```

To process the return value of this function, we can’t just access it in the reactor loop, because we never know when the Fiber got called and actually returns something.
This can be solved by just having a simple check like "if my_result != nil" in the reactor loop. This way, we can basically wait for the method to return something without blocking the reactor and therefore still allowing other operations to happen while we wait.
Ilya Grigorik has automated this behaviour within his "em-synchrony" library for the Eventmachine framework.

6 Asynchronous I/O Frameworks

There are numerous frameworks for programming languages that help programmers to abstract over the complexity of underlying OS functionality and keep the code focused on business logic.
This list is by no means complete, but it mentions some well known Frameworks for mainstream programming languages. I would strongly advice to look into these Frameworks before trying to implement an independent solution. Almost all of these Frameworks offer generic support for TCP/UDP connections that allow to implement any given protocol on top of an asynchronous base:

- Eventmachine: A Ruby Framework for Ruby (and JRuby). It provides clients for HTTP, Memcached, SMTP, Socks and numerous other protocols. It supports epoll (Linux), kqueue (BSD/OS X) and /dev/poll (Solaris)

- Twisted: A Python Framework. Basis for lots of Projects implementing HTTP, NNTP, IRC, FTP and others.
- Node.js: An evented I/O Framework for serverside Javascript. Node.js is using Google's V8 Javascript Engine.
- Apache Java, MINA: Allows programmers to create their own asynchronous servers. Shipping with SSL support.
- Netty: Java, Supposedly faster than MINA. It is the underlying basis of JXTA.

7 When to use it

While asynchronous I/O is a great tool for deployments that are I/O intensive, it sometimes isn't worth the additional complexity in code. Just doing a simple test with HTTP libraries that offer easy access to evented I/O operations shows that the total time actually waiting for a thread to start, execute and terminate itself again is really close to what the asynchronous libraries do with, in this case, epoll. Just as a small proof of concept, I used a benchmark script to run requests to a mix of large sites (google, yahoo, ...) and a site that responded slowly (simple web-application that just uses a sleep() command before answering).
The Ruby script used 3 different approaches:

- em-http: A HTTP library for Eventmachine. Eventmachine is an asynchronous I/O framework using the reactor pattern. The reactor is written in C to get arround the threading constraints imposed by the Ruby VM
- typhoeus: A library that has bindings to libcurl and is able to use libcurl's "multi' interface which internally relies on epoll for evented I/O
- threaded net::http: Ruby's default HTTP library. Since There is no way to request multiple URLs at once, we just launch a new thread for each request.

After running each script 20 times, these are the averaged results:

em-http	2.66 s
typhoeus	2.92 s
net::http	2.65 s

We can see that, in this case, the three approaches don't differ from each other by more than the margin of error in this quick experiment. For most "simple" operations with only 50-100 connections, the overhead created by the reactor is comparable to the overhead that regular threads create in this situation.

8 Conclusion

Asynchronous I/O is a powerful tool to help applications that have to handle large amounts of parallel I/O operations scale. While it is also possible to use it for generic applications with lesser requirements, it usually results in overly complex code without a lot of gained performance for the specific task. While there are frameworks and techniques that make the necessary code look cleaner, it might be best to have at least some experience with either functional programming or event-driven systems to be able to maintain a codebase that uses evented I/O.

References

[1] "Comparing Two High-Performance I/O Design Patterns"
Alexander Libman and Vladimir Gilbourd
2005
http://www.artima.com/articles/io_design_patterns.html

[2] Finding parallelism - How to survive in a multi-core world
Kai Jäger
2008
http://kaijaeger.com/publications/finding-parallelism-how-to-survive-in-a-multi-core-world.pdf

[3] Untangling Evented Code with Ruby Fibers
Ilya Grigorik
2010
http://www.igvita.com/2010/03/22/untangling-evented-code-with-ruby-fibers/